VISION

To be an illustrious Institution imparting quality engineering education and prepare globally accredited and socially responsible professionals

MISSION

M1. To establish state-of-the-art facilities and a conducive environment for education and research

M2. To collaborate with academia and industry

M3. To foster Professional Ethics, Innovation and Entrepreneurship

M4. To fulfil social obligations

DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND DATA SCIENCE

VISION

To become a prominent education and research center producing globally competent engineers in artificial intelligence and data science

MISSION

M1. To provide state-of-the-art facilities for quality education and research

M2. To interact with industries and institutions of higher learning

M3. To facilitate research activities and promote entrepreneurship

M4. To nurture ethics and responsible citizenship

PROGRAM EDUCATIONAL OBJECTIVES

Our graduates in a span of 3 to 5 years after graduation shall:

PEO1. Serve as successful Artificial Intelligence Engineers or Data Scientists professionals

PEO2. Pursue higher learning and/or take up business

PEO3. Exhibit honorable professionalism and social responsibility

DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND DATA SCIENCE

PROGRAM OUTCOMES (POs)

1. **Engineering knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
2. **Problem analysis:** Identify, formulate, review research literature, and analyse complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3. **Design/development of solutions:** Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
4. **Conduct investigations of complex problems:** Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
5. **Modern tool usage:** Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modelling to complex engineering activities with an understanding of the limitations.
6. **The engineer and society:** Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
7. **Environment and sustainability:** Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
8. **Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
9. **Individual and team work:** Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
10. **Communication:** Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
11. **Project management and finance:** Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
12. **Life-long learning:** Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change

PROGRAM SPECIFIC OUTCOMES

PSO1. Analyse and design solutions to real world problems using the knowledge of artificial intelligence and data science

PSO2. Build intelligent systems using appropriate hardware/software tools and techniques

DESIGN AND ANALYSIS OF ALGORITHMS			
Course Code	**21CS42**	CIE Marks	50
Teaching Hours/Week (L:T:P: S)	3:0:2:0	SEE Marks	50
Total Hours of Pedagogy	40 T + 20 P	Total Marks	100
Credits	04	Exam Hours	03

Course Learning Objectives:

CLO 1. Explain the methods of analyzing the algorithms and to analyze performance of algorithms.
CLO 2. State algorithm's efficiencies using asymptotic notations.
CLO 3. Solve problems using algorithm design methods such as the brute force method, greedy method, divide and conquer, decrease and conquer, transform and conquer, dynamic programming, backtracking and branch and bound.
CLO 4. Choose the appropriate data structure and algorithm design method for a specified application.
CLO 5. Introduce P and NP classes.

Teaching-Learning Process (General Instructions)

These are sample Strategies; which teachers can use to accelerate the attainment of the various course outcomes.

1. Lecturer method (L) does not mean only traditional lecture method, but different type of teaching methods may be adopted to develop the outcomes.
2. Show Video/animation films to explain functioning of various concepts.
3. Encourage collaborative (Group Learning) Learning in the class.
4. Ask at least three HOT (Higher order Thinking) questions in the class, which promotes critical thinking.
5. Adopt Problem Based Learning (PBL), which fosters students' Analytical skills, develop thinking skills such as the ability to evaluate, generalize, and analyze information rather than simply recall it.
6. Topics will be introduced in a multiple representation.
7. Show the different ways to solve the same problem and encourage the students to come up with their own creative ways to solve them.
8. Discuss how every concept can be applied to the real world - and when that's possible, it helps improve the students' understanding.

Module-1

Introduction: What is an Algorithm? It's Properties. Algorithm Specification-using natural language, using Pseudo code convention, Fundamentals of Algorithmic Problem solving, Analysis Framework-Time efficiency and space efficiency, Worst-case, Best-case and Average case efficiency.

Performance Analysis: Estimating Space complexity and Time complexity of algorithms.

Asymptotic Notations: Big-Oh notation (O), Omega notation (Ω), Theta notation(Θ) with examples, Basic efficiency classes, Mathematical analysis of Non-Recursive and Recursive Algorithms with Examples.

Brute force design technique: Selection sort, sequential search, string matching algorithm with complexity Analysis.

Textbook 1: Chapter 1 (Sections 1.1,1.2), Chapter 2(Sections 2.1,2.2,2.3,2.4), Chapter 3(Section 3.1,3.2)

Textbook 2: Chapter 1(section 1.1,1.2,1.3)

Laboratory Component:

1. Sort a given set of n integer elements using Selection Sort method and compute its time complexity. Run the program for varied values of n> 5000 and record the time taken to sort. Plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator. Demonstrate using C++/Java how the brute force method works along with its time complexity analysis: worst case, average case and best case.

Teaching-Learning Process	1. Problem based Learning. 2. Chalk & board, Active Learning. 3. Laboratory Demonstration.

Module-2

Divide and Conquer: General method, Recurrence equation for divide and conquer, solving it using Master's theorem. , Divide and Conquer algorithms and complexity Analysis of Finding the maximum & minimum, Binary search, Merge sort, Quick sort.

Decrease and Conquer Approach: Introduction, Insertion sort, Graph searching algorithms, Topological Sorting. It's efficiency analysis.

Textbook 2: Chapter 3(Sections 3.1,3.3,3.4,3.5,3.6)

Textbook 1: Chapter 4 (Sections 4.1,4.2,4.3), Chapter 5(Section 5.1,5.2,5.3)

Laboratory Component:

1. Sort a given set of n integer elements using Quick Sort method and compute its time

   complexity. Run the program for varied values of n> 5000 and record the time taken to sort. Plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator. Demonstrate using C++/Java how the divide-and-conquer method works along with its time complexity analysis: worst case, average case and best case.

2. Sort a given set of n integer elements using Merge Sort method and compute its time complexity. Run the program for varied values of n> 5000, and record the time taken to sort. Plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator. Demonstrate using C++/Java how the divide-and-conquer method works along with its time complexity analysis: worst case, average case and best case.

Teaching-Learning Process	1. Chalk & board, Active Learning, MOOC, Problem based Learning. 2. Laboratory Demonstration.

Module-3

Greedy Method: General method, Coin Change Problem, Knapsack Problem, solving Job sequencing with deadlines Problems.

Minimum cost spanning trees: Prim's Algorithm, Kruskal's Algorithm with performance analysis.

Single source shortest paths: Dijkstra's Algorithm.

Optimal Tree problem: Huffman Trees and Codes.

Transform and Conquer Approach: Introduction, Heaps and Heap Sort.

Textbook 2: Chapter 4(Sections 4.1,4.3,4.5)

Textbook 1: Chapter 9(Section 9.1,9.2,9.3,9.4), Chapter 6(section 6.4)

Laboratory Component:

Write & Execute C++/Java Program

1. To solve Knapsack problem using Greedy method.
2. To find shortest paths to other vertices from a given vertex in a weighted connected graph, using Dijkstra's algorithm.
3. To find Minimum Cost Spanning Tree of a given connected undirected graph using Kruskal's algorithm. Use Union-Find algorithms in your program.
4. To find Minimum Cost Spanning Tree of a given connected undirected graph using Prim's algorithm.

Teaching-Learning Process	1. Chalk & board, Active Learning, MOOC, Problem based Learning. 2. Laboratory Demonstration.

Module-4

Dynamic Programming: General method with Examples, Multistage Graphs.

Transitive Closure: Warshall's Algorithm. **All Pairs Shortest Paths**: Floyd's Algorithm,

Knapsack problem, Bellman-Ford Algorithm, Travelling Sales Person problem.

Space-Time Tradeoffs: Introduction, sorting by Counting, Input Enhancement in String Matching-Harspool's algorithm.

Textbook 2: Chapter 5 (Sections 5.1,5.2,5.4,5.9)

Textbook 1: Chapter 8(Sections 8.2,8.4), Chapter 7 (Sections 7.1,7.2)

Laboratory Component:

Write C++/ Java programs to

1. Solve All-Pairs Shortest Paths problem using Floyd's algorithm.
2. Solve Travelling Sales Person problem using Dynamic programming.
3. Solve 0/1 Knapsack problem using Dynamic Programming method.

Teaching-Learning Process	1. Chalk & board, Active Learning, MOOC, Problem based Learning. 2. Laboratory Demonstration.

Module-5

Backtracking: General method, solution using back tracking to N-Queens problem, Sum of subsets problem, Graph coloring, Hamiltonian cycles Problems.

Branch and Bound: Assignment Problem, Travelling Sales Person problem, 0/1 Knapsack problem

NP-Complete and NP-Hard problems: Basic concepts, non- deterministic algorithms, P, NP, NP-Complete, and NP-Hard classes.

Textbook 1: Chapter 12 (Sections 12.1,12.2) Chapter 11(11.3)

Textbook 2: Chapter 7 (Sections 7.1,7.2,7.3,7.4,7.5) Chapter 11 (Section 11.1)

Laboratory Component:

1. Design and implement C++/Java Program to find a subset of a given set S = {Sl, S2,…, Sn} of n positive integers whose SUM is equal to a given positive integer d. For example, if S = {1, 2, 5, 6, 8} and d= 9, there are two solutions {1, 2, 6} and {1, 8}. Display a suitable message, if the given problem instance doesn't have a solution.
2. Design and implement C++/Java Program to find all Hamiltonian Cycles in a connected undirected Graph G of n vertices using backtracking principle.

Teaching-Learning Process	1. Chalk & board, Active Learning, MOOC, Problem based learning. 2. Laboratory Demonstration.

Course outcome (Course Skill Set)

At the end of the course the student will be able to:

CO 1. Analyze the performance of the algorithms, state the efficiency using asymptotic notations and analyze mathematically the complexity of the algorithm.
CO 2. Apply divide and conquer approaches and decrease and conquer approaches in solving the problems analyze the same
CO 3. Apply the appropriate algorithmic design technique like greedy method, transform and conquer approaches and compare the efficiency of algorithms to solve the given problem.
CO 4. Apply and analyze dynamic programming approaches to solve some problems. and improve an algorithm time efficiency by sacrificing space.
CO 5. Apply and analyze backtracking, branch and bound methods and to describe P, NP and NP-Complete problems.

Assessment Details (both CIE and SEE)

The weightage of Continuous Internal Evaluation (CIE) is 50% and for Semester End Exam (SEE) is 50%. The minimum passing mark for the CIE is 40% of the maximum marks (20 marks). A student shall be deemed to have satisfied the academic requirements and earned the credits allotted to each subject/ course if the student secures not less than 35% (18 Marks out of 50) in the semester-end examination (SEE), and a minimum of 40% (40 marks out of 100) in the sum total of the CIE (Continuous Internal Evaluation) and SEE (Semester End Examination) taken together

Continuous Internal Evaluation:

Three Unit Tests each of **20 Marks (duration 01 hour**)

- First test at the end of 5th week of the semester
- Second test at the end of the 10th week of the semester
- Third test at the end of the 15th week of the semester

Two assignments each of **10 Marks**

- First assignment at the end of 4th week of the semester
- Second assignment at the end of 9th week of the semester

Practical Sessions need to be assessed by appropriate rubrics and viva-voce method. This will contribute to **20 marks**.

- Rubrics for each Experiment taken average for all Lab components – 15 Marks.
- Viva-Voce– 5 Marks (more emphasized on demonstration topics)

The sum of three tests, two assignments, and practical sessions will be out of 100 marks and will be **scaled down to 50 marks**
(to have a less stressed CIE, the portion of the syllabus should not be common /repeated for any of the methods of the CIE. Each method of CIE should have a different syllabus portion of the course).

CIE methods /question paper has to be designed to attain the different levels of Bloom's taxonomy as per the outcome defined for the course.

Semester End Examination:

Theory SEE will be conducted by University as per the scheduled timetable, with common question papers for the subject (**duration 03 hours)**

1. The question paper will have ten questions. Each question is set for 20 marks. Marks scored shall be proportionally reduced to 50 marks
2. There will be 2 questions from each module. Each of the two questions under a module (with a maximum of 3 sub-questions), **should have a mix of topics** under that module.

The students have to answer 5 full questions, selecting one full question from each module

Suggested Learning Resources:

Textbooks

1. Introduction to the Design and Analysis of Algorithms, Anany Levitin: 2nd Edition, 2009. Pearson.
2. Computer Algorithms/C++, Ellis Horowitz, SatrajSahni and Rajasekaran, 2nd Edition, 2014, Universities Press.

Reference Books

1. Introduction to Algorithms, Thomas H. Cormen, Charles E. Leiserson, Ronal L. Rivest, Clifford Stein, 3rd Edition, PHI.
2. Design and Analysis of Algorithms, S. Sridhar, Oxford (Higher Education)

Web links and Video Lectures (e-Resources):

1. http://elearning.vtu.ac.in/econtent/courses/video/CSE/06CS43.html
2. https://nptel.ac.in/courses/106/101/106101060/
3. http://elearning.vtu.ac.in/econtent/courses/video/FEP/ADA.html
4. http://cse01-iiith.vlabs.ac.in/
5. http://openclassroom.stanford.edu/MainFolder/CoursePage.php?course=IntroToAlgorithms

Activity Based Learning (Suggested Activities in Class)/ Practical Based learning

1. Real world problem solving and puzzles using group discussion. E.g., Fake coin identification, Peasant, wolf, goat, cabbage puzzle, Konigsberg bridge puzzle etc.,
2. Demonstration of solution to a problem through programming.

Module-1

Introduction: What is an Algorithm?

An ***algorithm*** is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time.

This definition can be illustrated by a simple diagram (Figure 1.1).

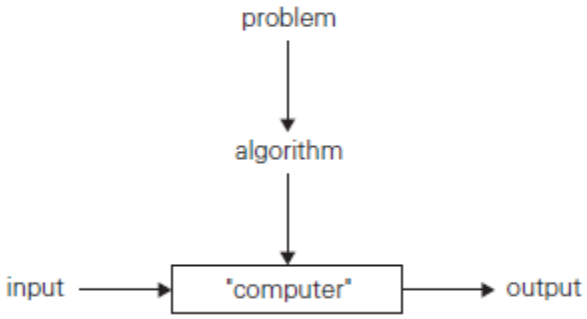


Figure: The notion of the algorithm.

In order for some instructions to be an algorithm, it must have the following characteristics:

- The nonambiguity requirement for each step of an algorithm cannot be compromised.
- The range of inputs for which an algorithm works has to be specified carefully.
- The same algorithm can be represented in several different ways.
- There may exist several algorithms for solving the same problem.
- Algorithms for the same problem can be based on very different ideas and can solve the problem with dramatically different speeds.

Properties of Algorithm:

- It should terminate after a finite time.
- It should produce at least one output.
- It should take zero or more input.
- It should be deterministic means giving the same output for the same input case.
- Every step in the algorithm must be effective i.e. every step should do some work.

Algorithm Specification:

We can depict an algorithm in many ways.

- **Natural language:** Implement a natural language like English
- **Flow charts:** ***Flowchart***, a method of expressing an algorithm by a collection of connected geometric shapes containing descriptions of the algorithm's steps. This representation technique

has proved to be inconvenient for all but very simple algorithms; nowadays, it can be found only in old algorithm books.

- **Pseudo code:** ***Pseudocode*** is a mixture of a natural language and programming language-like constructs.

Euclid's algorithm:

Euclid's algorithm is based on applying repeatedly the equality

$$\gcd(m, n) = \gcd(n, m \bmod n),$$

For example, gcd *(*60*,* 24*)* can be computed as follows:

$$\gcd(60, 24) = \gcd(24, 12) = \gcd(12, 0) = 12.$$

Euclid's algorithm for computing gcd*(m, n)*

Step 1 If $n = 0$, return the value of m as the answer and stop; otherwise, proceed to Step 2.

Step 2 Divide m by n and assign the value of the remainder to r.

Step 3 Assign the value of n to m and the value of r to n. Go to Step 1.

Alternatively, we can express the same algorithm in pseudocode:

ALGORITHM *Euclid (m, n)*

//Computes gcd *(m, n)* by Euclid's algorithm

//Input: Two nonnegative, not-both-zero integers m and n

//Output: Greatest common divisor of m and n

while $n \neq 0$ **do**

$r \leftarrow m \bmod n$

$m \leftarrow n$

$n \leftarrow r$

return m

Consecutive integer checking algorithm for computing gcd*(m, n)*

Step 1 Assign the value of min $\{m, n\}$ to t.

Step 2 Divide m by t. If the remainder of this division is 0, go to Step 3; otherwise, go to Step 4.

Step 3 Divide n by t. If the remainder of this division is 0, return the value of t as the answer and stop; otherwise, proceed to Step 4.

Step 4 Decrease the value of t by 1. Go to Step 2.

Middle-school procedure for computing gcd *(m, n)*

Step 1 Find the prime factors of m.

Step 2 Find the prime factors of n.

Step 3 Identify all the common factors in the two prime expansions found in Step 1 and Step 2. (If p is a common factor occurring pm and pn times in m and n, respectively, it should be repeated min $\{pm, pn\}$ times.)

Step 4 Compute the product of all the common factors and return it as the greatest common divisor of the numbers given.

Thus, for the numbers 60 and 24, we get

$60 = 2 \cdot 2 \cdot 3 \cdot 5$

$24 = 2 \cdot 2 \cdot 2 \cdot 3$

$\gcd(60, 24) = 2 \cdot 2 \cdot 3 = 12$.

Algorithm for generating consecutive primes not exceeding any given integer $n > 1$. (*sieve of Eratosthenes*)

ALGORITHM *Sieve*(n)

```
//Implements the sieve of Eratosthenes
//Input: A positive integer n > 1
//Output: Array L of all prime numbers less than or equal to n

for p ← 2 to n do A[p] ← p
for p ← 2 to ⌊√n⌋ do    //see note before pseudocode
    if A[p] ≠ 0          //p hasn't been eliminated on previous passes
        j ← p * p
        while j ≤ n do
            A[j] ← 0     //mark element as eliminated
            j ← j + p
//copy the remaining elements of A to array L of the primes
i ← 0
for p ← 2 to n do
    if A[p] ≠ 0
        L[i] ← A[p]
        i ← i + 1
return L
```

As an example, consider the application of the algorithm to finding the list of primes not exceeding $n = 25$:

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
2	3		5		7		9		11		13		15		17		19		21		23		25
2	**3**		5		7				11		13				17		19				23		25
2	3		**5**		7				11		13				17		19				23		

Fundamentals of Algorithmic Problem solving:

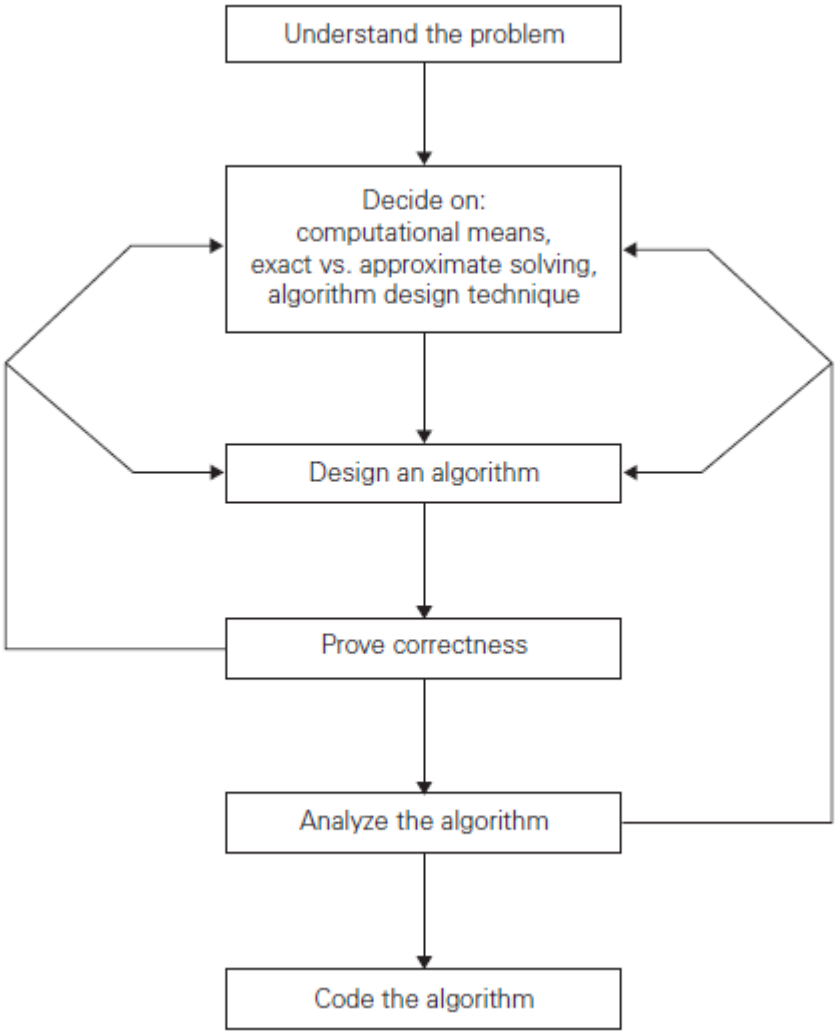


Figure: Algorithm design and analysis process.

Understanding the Problem: From a practical perspective, the first thing you need to do before designing an algorithm is to understand completely the problem given. Read the problem's description carefully and ask questions if you have any doubts about the problem, do a few small examples by hand, think about special cases, and ask questions again if needed.

An input to an algorithm specifies an instance of the problem the algorithm solves. It is very important to specify exactly the set of instances the algorithm needs to handle. (As an example, recall the variations in the set of instances for the three greatest common divisor algorithms discussed in the previous section.) If you fail to do this, your algorithm may work correctly for a majority of inputs but crash on some "boundary" value. Remember that a correct algorithm is not one that works most of the time, but one that works correctly for all legitimate inputs.

Do not skip on this first step of the algorithmic problem-solving process; otherwise, you will run the risk of unnecessary rework.

Ascertaining the Capabilities of the Computational Device: Once you completely understand a problem, you need to ascertain the capabilities of the computational device the algorithm is intended for. The vast majority of algorithms in use today are still destined to be programmed for a computer closely resembling the von Neumann machine: A computer architecture outlined by the prominent Hungarian-American mathematician John von Neumann (1903– 1957), in collaboration with A. Burks and H. Goldstine, in 1946. The essence of this architecture is captured by the so-called random-access machine (RAM). Its central assumption is that instructions are executed one after another, one operation at a time. Accordingly, algorithms designed to be executed on such machines are called sequential algorithms.

The central assumption of the RAM model does not hold for some newer computers that can execute operations concurrently, i.e., in parallel. Algorithms that take advantage of this capability are called parallel algorithms. Still, studying the classic techniques for design and analysis of algorithms under the RAM model remains the cornerstone of algorithmic for the foreseeable future.

Algorithm Design Techniques: An ***algorithm design technique*** (or "strategy" or "paradigm") is a general approach to solving problems algorithmically that is applicable to a variety of problems from different areas of computing.

Designing an Algorithm and Data Structures: One should pay close attention to choosing data structures appropriate for the operations performed by the algorithm. For example, the sieve of Eratosthenes introduced in Section 1.1 would run longer if we used a linked list instead of an array in its implementation.

Algorithms + Data Structures = Programs

In the new world of object-oriented programming, data structures remain crucially important for both design and analysis of algorithms.

Methods of Specifying an Algorithm: Once you have designed an algorithm, you need to specify it in some fashion. In Section 1.1, to give you an example, Euclid's algorithm is described in words (in a free and also a step-by-step form) and in pseudocode. These are the two options that are most widely used nowadays for specifying algorithms.

Using a ***natural language*** has an obvious appeal; however, the inherent ambiguity of any natural language makes a succinct and clear description of algorithms surprisingly difficult. Nevertheless, being able to do this is an important skill that you should strive to develop in the process of learning algorithms.

Pseudocode is a mixture of a natural language and programming language-like constructs. Pseudocode is usually more precise than natural language, and its usage often yields more succinct algorithm descriptions.

In the earlier days of computing, the dominant vehicle for specifying algorithms was a ***flowchart***, a method of expressing an algorithm by a collection of connected geometric shapes containing descriptions of the algorithm's steps. This representation technique has proved to be inconvenient for all but very simple algorithms; nowadays, it can be found only in old algorithm books.

Proving an Algorithm's Correctness: Once an algorithm has been specified, you have to prove its ***correctness***. That is, you have to prove that the algorithm yields a required result for every legitimate input in a finite amount of time.

For some algorithms, a proof of correctness is quite easy; for others, it can be quite complex. A common technique for proving correctness is to use mathematical induction because an algorithm's iterations provide a natural sequence of steps needed for such proofs. It might be worth mentioning that although tracing the algorithm's performance for a few specific inputs can be a very worthwhile activity, it cannot prove the algorithm's correctness conclusively. But in order to show that an algorithm is incorrect, you need just one instance of its input for which the algorithm fails.

The notion of correctness for approximation algorithms is less straightforward than it is for exact algorithms. For an approximation algorithm, we usually would like to be able to show that the error produced by the algorithm does not exceed a predefined limit.

Analyzing an Algorithm: We usually want our algorithms to possess several qualities. After correctness, by far the most important is ***efficiency***. In fact, there are two kinds of algorithm efficiency: ***time efficiency***, indicating how fast the algorithm runs, and ***space efficiency***, indicating how much extra memory it uses.

Coding an Algorithm: Most algorithms are destined to be ultimately implemented as computer pro-grams. Programming an algorithm presents both a peril and an opportunity. The peril lies in the possibility of making the transition from an algorithm to a pro-gram either incorrectly or very inefficiently. Some influential computer scientists strongly believe that unless the correctness of a computer program is proven with full mathematical rigor, the program cannot be considered correct.

As a practical matter, the validity of programs is still established by testing. Testing of computer programs is an art rather than a science, but that does not mean that there is nothing in it to learn. Look up books devoted to testing and debugging; even more important, test and debug your program thoroughly whenever you implement an algorithm.

We assume that inputs to algorithms belong to the specified sets and hence require no verification. When implementing algorithms as programs to be used in actual applications, you should provide such verifications.

Analysis Framework:

Analysis of algorithm is investigation of an algorithm's efficiency with respect to two resources: running time and memory space. There are two kinds of efficiency:

Time efficiency: ***Time efficiency***, also called ***time complexity***, indicates how fast an algorithm in question runs.

Space efficiency: ***Space efficiency***, also called ***space complexity***, refers to the amount of memory units required by the algorithm in addition to the space needed for its input and output.

1. Measuring an Input's Size
2. Units for Measuring Running Time
3. Orders of Growth
4. Worst-Case, Best-Case, and Average-Case Efficiencies

Measuring an Input's Size:

- An algorithm's efficiency is expressed as a function of some parameter *n* indicating the algorithm's input size. (Some algorithms require more than one parameter to indicate the size of their inputs).
- *n* indicates the size of the list for problems of sorting, searching, finding the list'ssmallest element etc.
- In evaluating of a polynomial, $p(x) = a_n x_n + a_{n-1} x_{n-1} + \ldots + a_0$ *n* indicates the polynomial's degree or the number of its coefficients, which is larger by one thanits degree.
- In computing the product of two *n*-by-*n* matrices, there are two natural measuresof size, one is the matrix order *n*, and the other is the total number of element *N* in the matrices being multiplied.
- The choice of an appropriate size metric can be influenced by operations of the algorithm in question. Eg., for a spell-checking algorithm, the number of characters must be measured if the algorithm examines individual characters; if it works by processing words, their number of words must be considered.
- For algorithms involving properties of numbers, input size is measured by the number of bits in the *n*'s binary representation:

$$b = \log_2 n + 1$$

Units for Measuring Running Time:

Basic Operation:

- The most important operation of the algorithm, called the basic operation, which is the operation contributing the most to the total running time is identified, and the number of times the basic operation is executed is computed.
- Usually the basic operation is the algorithm's innermost loop. E.g., most sorting algorithms work by comparing elements. Therefore, basic operation would be key comparison.
- Therefore, time efficiency is measured by counting the number of times an algorithm's basic operation is executed on inputs of size n.

Orders of Growth:

- Efficiency analysis framework ignores multiplicative constants and concentrates on the basic operation count's order of growth to within a constant multiple for large-size inputs.
- A difference in running times on small inputs is not what really distinguishes efficient algorithms from inefficient ones. For large values of n, it is the function's order of growth that counts.

Values (some approximate) of several functions important for analysis of algorithms

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

- The function growing the slowest among these is the logarithmic function
- Exponential function 2^n and the factorial function n! grow so fast that theirvalues become astronomically large even for rather small values of *n*. They are referred to as "exponential-growth functions".
- Algorithms that require an exponential number of operations are practical for solving only problems of very small sizes.
- For a twofold increase in the value of *n*,
  $\log_2 n$ increases just by 1. ($\log_2 2n = \log_2 2 + \log_2 n = 1 + \log_2 n$)
  Linear function increases twofold.

n-log-n increases slightly more that twofold. Quadratic function n^2 increases fourfold ($(2n)^2 = 4n^2$)Cubic function n^3 increases eightfold. $(2n)^3 = 8n^3$

The value of 2^n is squared ($(2^n)^2$) and n! increases much more.

Worst-Case, Best-Case and Average-Case Efficiencies:

For many algorithms, running time depends not only on an input size but also on the specifics of a particular input. E.g., Consider sequential search.

ALGORITHM *SequentialSearch*$(A[0..n-1], K)$
//Searches for a given value in a given array by sequential search
//Input: An array $A[0..n-1]$ and a search key K
//Output: The index of the first element in A that matches K
// or -1 if there are no matching elements
$i \leftarrow 0$
while $i < n$ **and** $A[i] \neq K$ **do**
$i \leftarrow i + 1$
if $i < n$ **return** i
else return -1

Worst-Case Efficiency:

- The worst-case efficiency of an algorithm is its efficiency for the worst-case input of size n, for which the algorithm runs the longest.
- The algorithm is analysed to see what kind of inputs yield the largest value of the basic operation's count C(n) among all possible inputs of size n and then compute this worst-case value $C_{worst}(n)$.
- E.g., In sequential search, when there is no matching element or when the first matching element is the last one in the list, the algorithm makes the largest number of key comparisons,

$$\mathbf{C_{worst}(n) = n}$$

Best-Case Efficiency

- The best-case efficiency of an algorithm is its efficiency for the best-case input of size n, for which the algorithm runs the fastest.
- The kind of inputs for which the count C(n) will be the smallest among all possible inputs of size n is determined as the best-case efficiency.
- E.g., for sequential search, best-case inputs will be lists of size n with their first elements equal to search key; therefore, $\mathbf{C_{best}(n) = 1.}$

Average-case Efficiency

- Average case efficiency yields necessary information about an algorithm's behaviour on a "typical" or "random" input.
- To analyse the algorithm's average-case efficiency, some assumptions about possible inputs of size n must be made.
- Investigation of the average-case efficiency is considerable more difficult than the others.
- It cannot be obtained by taking the average of the worst-case and the best-case efficiencies.

Let's consider again sequential search. The standard assumptions are that (a) the probability of a successful search is equal to p $(0 \le p \le 1)$ and (b) the probability of the first match occurring in the ith position of the list is the same for every i. Under these assumptions—the validity of which is usually difficult to verify, their reasonableness notwithstanding—we can find the average number of key comparisons $C_{avg}(n)$ as follows. In the case of a successful search, the probability of the first match occurring in the ith position of the list is p/n for every i, and the number of comparisons made by the algorithm in such a situation is obviously i. In the case of an unsuccessful search, the number of comparisons will be n with the probability of such a search being $(1 - p)$. Therefore,

$$C_{avg}(n) = [1 \cdot \frac{p}{n} + 2 \cdot \frac{p}{n} + \cdots + i \cdot \frac{p}{n} + \cdots + n \cdot \frac{p}{n}] + n \cdot (1 - p)$$
$$= \frac{p}{n}[1 + 2 + \cdots + i + \cdots + n] + n(1 - p)$$
$$= \frac{p}{n}\frac{n(n+1)}{2} + n(1 - p) = \frac{p(n+1)}{2} + n(1 - p).$$

Performance Analysis:

There is a criterion for judging algorithms that have a direct relationship to performance. These have to do with their computing time and space required. Time and space directly links with the performance of your algorithm. So time and space complexity are two major criteria to consider.

Space complexity: The space complexity of an algorithm is the amount of memory it need to run for completion. i.e. how much space will your algorithm take between start and end of your algorithm. Your algorithm space is the sum of the following components:

1. A fixed part that is independent of the characteristics (e.g., number, size) of the inputs and outputs. This part typically includes the instruction space (i.e., space for the code), space for simple variables and fixed-size component variables (also called *aggregate*), space for constants, and so on.

2. A variable part that consists of the space needed by component variables whose size is dependent on the particular problem instance being solved, the space needed by referenced variables (to the extent that this depends on instance characteristics), and the recursion stack space (insofar as this space depends on the instance characteristics).

So The space requirement of your algorithms is

$$S(P)= C + S_p$$

C = constant

S_p = Instant and variable characteristics

```
1   Algorithm abc(a, b, c)
2   {
3       return  a + b + b * c + (a + b - c)/(a + b) + 4.0;
4   }
```

Algorithm 1.5 Computes $a + b + b * c + (a + b - c)/(a + b) + 4.0$

```
1   Algorithm Sum(a, n)
2   {
3       s := 0.0;
4       for i := 1 to n do
5           s := s + a[i];
6       return s;
7   }
```

Algorithm 1.6 Iterative function for sum

Example 1.5 The problem instances for Algorithm 1.6 are characterized by n, the number of elements to be summed. The space needed by n is one word, since it is of type *integer*. The space needed by a is the space needed by variables of type array of floating point numbers. This is at least n words, since a must be large enough to hold the n elements to be summed. So, we obtain $S_{\mathsf{Sum}}(n) \geq (n + 3)$ (n for $a[\,]$, one each for n, i, and s). □

Example 1.6 Let us consider the algorithm RSum (Algorithm 1.7). As in the case of Sum, the instances are characterized by n. The recursion stack space includes space for the formal parameters, the local variables, and the return address. Assume that the return address requires only one word of memory. Each call to RSum requires at least three words (including space for the values of n, the return address, and a pointer to $a[\,]$). Since the depth of recursion is $n + 1$, the recursion stack space needed is $\geq 3(n + 1)$. □

Time Complexity:

The time complexity of an algorithm is the amount of time it needs to run for completion. You can define the time complexity of any algorithms by counting the no. of program steps instead of determining the exact no. of operations.

A program step is defined as a syntactically or semantically meaning full segment of your program that has an execution time i.e. independent.

The no. of steps of any program statement depends on the kind of statement. For example, comments count as zero steps. And assignment statement (not involving any calls to other algorithms) is counted as one step. In an iterative statement such as "for"," while", "repeat", "until" statements, we consider the step count only for the control part of the statement.

Statement	s/e	frequency	total steps
1 **Algorithm** Sum(a, n)	0	–	0
2 {	0	–	0
3 $s := 0.0$;	1	1	1
4 **for** $i := 1$ **to** n **do**	1	$n+1$	$n+1$
5 $s := s + a[i]$;	1	n	n
6 **return** s;	1	1	1
7 }	0	–	0
Total			$2n+3$

Asymptotic Notations:

The efficiency analysis framework concentrates on the order of growth of an algorithm's basic operation count as the principal indicator of the algorithm's efficiency. To compare and rank such orders of growth, computer scientists use three notations: ***O* (big oh), Ω (big omega), and Θ (big theta).**

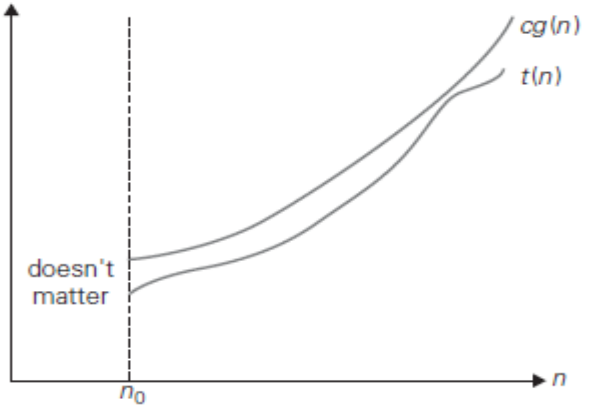
In the following discussion, *t (n)* and *g(n)* can be any nonnegative functions defined on the set of natural numbers. In the context we are interested in, *t (n)* will be an algorithm's running time (usually indicated by its basic operation count *C(n)*), and *g(n)* will be some simple function to compare the count with.

***O*-Notation:**

DEFINITION A function *t (n)* is said to be in *O(g(n))*, denoted *t (n)* ∈ *O(g(n)),* if *t (n)* is bounded above by some constant multiple of *g(n)* for all large *n,* i.e., if there exist some positive constant *c* and some nonnegative integer n_0 such that

$$t(n) \leq cg(n) \text{ for all } n \geq n_0.$$

The definition is illustrated in Figure below where, for the sake of visual clarity, *n* is extended to be a real number.



Big-oh notation: $t(n) \in O(g(n))$.

As an example, let us formally prove one of the assertions made in the introduction: $100n + 5 \in O(n^2)$. Indeed,

$$100n + 5 \leq 100n + n \text{ (for all } n \geq 5) = 101n \leq 101n^2.$$

Thus, as values of the constants c and n_0 required by the definition, we can take 101 and 5, respectively. Note that the definition gives us a lot of freedom in choosing specific values for constants c and n_0. For example, we could also reason that

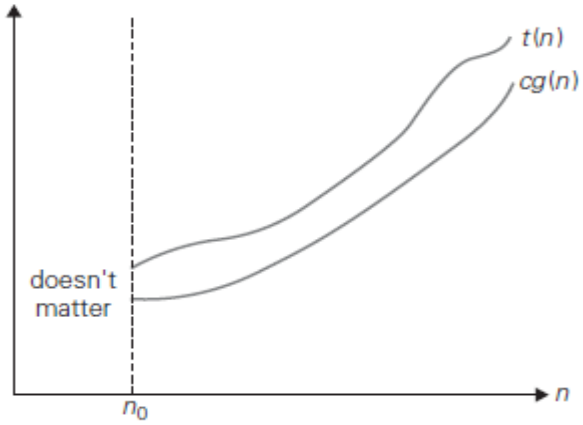
$$100n + 5 \leq 100n + 5n \text{ (for all } n \geq 1) = 105n$$

to complete the proof with $c = 105$ and $n_0 = 1$.

Ω (big omega):

DEFINITION A function *t (n)* is said to be in Ω *(g(n)),* denoted *t (n)* ∈ Ω *(g(n)),* if *t (n)* is bounded below by some positive constant multiple of *g(n)* for all large *n,* i.e., if there exist some positive constant *c* and some nonnegative integer n_0 such that

***t (n)* ≥ *cg(n)* for all *n* ≥ n_0.**



Big-omega notation: *t (n)* ∈Ω *(g(n))*.

Here is an example of the formal proof that $n^3 \in \Omega(n^2)$:

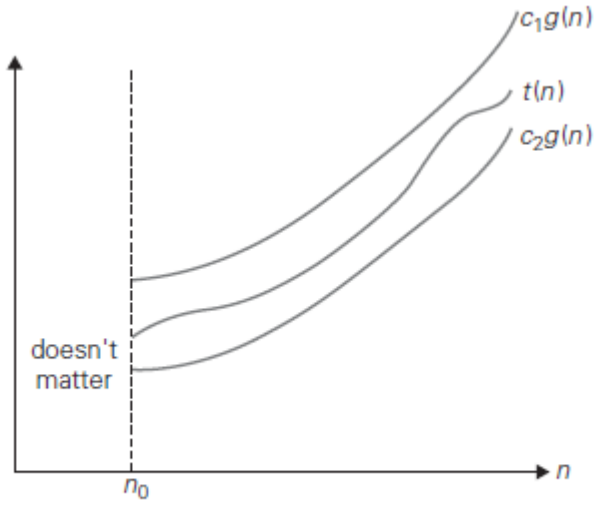
$$n^3 \geq n^2 \text{ for all } n \geq 0,$$

i.e., we can select $c = 1$ and $n_0 = 0$.

Θ (big theta):

DEFINITION A function $t(n)$ is said to be in $\Theta(g(n))$, denoted $t(n) \in \Theta(g(n))$, if $t(n)$ is bounded both above and below by some positive constant multiples of $g(n)$ for all large n, i.e., if there exist some positive constants c_1 and c_2 and some nonnegative integer n_0 such that

$$\boldsymbol{c_2 g(n) \leq t(n) \leq c_1 g(n) \text{ for all } n \geq n_0.}$$



Big-theta notation: $t(n) \in \Theta(g(n))$.

For example, let us prove that $\frac{1}{2}n(n-1) \in \Theta(n^2)$. First, we prove the right inequality (the upper bound):

$$\frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \leq \frac{1}{2}n^2 \quad \text{for all } n \geq 0.$$

Second, we prove the left inequality (the lower bound):

$$\frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \geq \frac{1}{2}n^2 - \frac{1}{2}n\frac{1}{2}n \text{ (for all } n \geq 2) = \frac{1}{4}n^2.$$

Hence, we can select $c_2 = \frac{1}{4}$, $c_1 = \frac{1}{2}$, and $n_0 = 2$.

Useful Property Involving the Asymptotic Notations:

THEOREM If $t_1(n) \in O(g_1(n))$ and $t_2(n) \in O(g_2(n))$, then

$$t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\}).$$

(The analogous assertions are true for the Ω and Θ notations as well.)

PROOF The proof extends to orders of growth the following simple fact about four arbitrary real numbers a_1, b_1, a_2, b_2: if $a_1 \le b_1$ and $a_2 \le b_2$, then $a_1 + a_2 \le 2 \max\{b_1, b_2\}$.

Since $t_1(n) \in O(g_1(n))$, there exist some positive constant c_1 and some non-negative integer n_1 such that

$$t_1(n) \le c_1 g_1(n) \quad \text{for all } n \ge n_1.$$

Similarly, since $t_2(n) \in O(g_2(n))$,

$$t_2(n) \le c_2 g_2(n) \quad \text{for all } n \ge n_2.$$

Let us denote $c_3 = \max\{c_1, c_2\}$ and consider $n \ge \max\{n_1, n_2\}$ so that we can use both inequalities. Adding them yields the following:

$$\begin{aligned} t_1(n) + t_2(n) &\le c_1 g_1(n) + c_2 g_2(n) \\ &\le c_3 g_1(n) + c_3 g_2(n) = c_3[g_1(n) + g_2(n)] \\ &\le c_3 2 \max\{g_1(n), g_2(n)\}. \end{aligned}$$

Hence, $t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\})$, with the constants c and n_0 required by the O definition being $2c_3 = 2 \max\{c_1, c_2\}$ and $\max\{n_1, n_2\}$, respectively. ■

Using Limits for Comparing Orders of Growth:

Though the formal definitions of O, Ω, and Θ are indispensable for proving their abstract properties, they are rarely used for comparing the orders of growth of two specific functions.

A much more convenient method for doing so is based on computing the limit of the ratio of two functions in question. Three principal cases may arise:

$$\lim_{n\to\infty} \frac{t(n)}{g(n)} = \begin{cases} 0 & \text{implies that } t(n) \text{ has a smaller order of growth than } g(n), \\ c & \text{implies that } t(n) \text{ has the same order of growth as } g(n), \\ \infty & \text{implies that } t(n) \text{ has a larger order of growth than } g(n).^3 \end{cases}$$

Note that the first two cases mean that $t(n) \in O(g(n))$, the last two mean that $t(n) \in \Omega(g(n))$, and the second case means that $t(n) \in \Theta(g(n))$.

The limit-based approach is often more convenient than the one based on the definitions because it can take advantage of the powerful calculus techniques developed for computing limits, such as L'Hôpital's rule

$$\lim_{n\to\infty} \frac{t(n)}{g(n)} = \lim_{n\to\infty} \frac{t'(n)}{g'(n)}$$

and Stirling's formula

$$n! \approx \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \quad \text{for large values of } n.$$

EXAMPLE 1 Compare the orders of growth of $\frac{1}{2}n(n-1)$ and n^2. (This is one of the examples we used at the beginning of this section to illustrate the definitions.)

$$\lim_{n\to\infty}\frac{\frac{1}{2}n(n-1)}{n^2}=\frac{1}{2}\lim_{n\to\infty}\frac{n^2-n}{n^2}=\frac{1}{2}\lim_{n\to\infty}(1-\frac{1}{n})=\frac{1}{2}.$$

Since the limit is equal to a positive constant, the functions have the same order of growth or, symbolically, $\frac{1}{2}n(n-1)\in\Theta(n^2)$. ■

EXAMPLE 2 Compare the orders of growth of $\log_2 n$ and $\sqrt{n}$. (Unlike Example 1, the answer here is not immediately obvious.)

$$\lim_{n\to\infty}\frac{\log_2 n}{\sqrt{n}}=\lim_{n\to\infty}\frac{(\log_2 n)'}{(\sqrt{n})'}=\lim_{n\to\infty}\frac{(\log_2 e)\frac{1}{n}}{\frac{1}{2\sqrt{n}}}=2\log_2 e\lim_{n\to\infty}\frac{1}{\sqrt{n}}=0.$$

Since the limit is equal to zero, $\log_2 n$ has a smaller order of growth than $\sqrt{n}$. (Since $\lim_{n\to\infty}\frac{\log_2 n}{\sqrt{n}}=0$, we can use the so-called ***little-oh notation***: $\log_2 n\in o(\sqrt{n})$. Unlike the big-Oh, the little-oh notation is rarely used in analysis of algorithms.)

EXAMPLE 3 Compare the orders of growth of $n!$ and 2^n.
Taking advantage of Stirling's formula, we get

$$\lim_{n\to\infty}\frac{n!}{2^n}=\lim_{n\to\infty}\frac{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n}{2^n}=\lim_{n\to\infty}\sqrt{2\pi n}\frac{n^n}{2^n e^n}=\lim_{n\to\infty}\sqrt{2\pi n}\left(\frac{n}{2e}\right)^n=\infty.$$

Therefore, $n!$ grows faster than 2^n and hence $n!\in\Omega(2^n)$.

Basic Efficiency Classes:

Even though the efficiency analysis framework puts together all the functions whose orders of growth differ by a constant multiple, there are still infinitely many such classes. (For example, the exponential functions a^n have different orders of growth for different values of base a.) Therefore, it may come as a surprise that the time efficiencies of a large number of algorithms fall into only a few classes. These classes are listed in Table below.

Basic asymptotic efficiency classes

Class	Name	Comments
1	*constant*	Short of best-case efficiencies, very few reasonable examples can be given since an algorithm's running time typically goes to infinity when its input size grows infinitely large.
$\log n$	*logarithmic*	Typically, a result of cutting a problem's size by a constant factor on each iteration of the algorithm (see Section 4.4). Note that a logarithmic algorithm cannot take into account all its input or even a fixed fraction of it: any algorithm that does so will have at least linear running time.
n	*linear*	Algorithms that scan a list of size n (e.g., sequential search) belong to this class.
$n \log n$	*linearithmic*	Many divide-and-conquer algorithms (see Chapter 5), including mergesort and quicksort in the average case, fall into this category.
n^2	*quadratic*	Typically, characterizes efficiency of algorithms with two embedded loops (see the next section). Elementary sorting algorithms and certain operations on $n \times n$ matrices are standard examples.
n^3	*cubic*	Typically, characterizes efficiency of algorithms with three embedded loops (see the next section). Several nontrivial algorithms from linear algebra fall into this class.
2^n	*exponential*	Typical for algorithms that generate all subsets of an n-element set. Often, the term "exponential" is used in a broader sense to include this and larger orders of growth as well.
$n!$	*factorial*	Typical for algorithms that generate all permutations of an n-element set.

Mathematical Analysis of Non-Recursive Algorithms:

General Plan for Analysing the Time Efficiency of Non-Recursive Algorithms

1. Decide on a parameter (or parameters) indicating an input's size.
2. Identify the algorithm's basic operation. (As a rule, it is located in the innermost loop.)
3. Check whether the number of times the basic operation is executed depends only on the size of an input. If it also depends on some additional property, the worst-case, average-case, and, if necessary, best-case efficiencies have to be investigated separately.
4. Set up a sum expressing the number of times the algorithm's basic operation is executed.
5. Using standard formulas and rules of sum manipulation, either find a closed form formula for the count or, at the very least, establish its order of growth.

In particular, we use especially frequently two basic rules of sum manipulation

$$\sum_{i=l}^{u} ca_i = c\sum_{i=l}^{u} a_i, \qquad \textbf{(R1)}$$

$$\sum_{i=l}^{u} (a_i \pm b_i) = \sum_{i=l}^{u} a_i \pm \sum_{i=l}^{u} b_i, \qquad \textbf{(R2)}$$

and two summation formulas

$$\sum_{i=l}^{u} 1 = u - l + 1 \quad \text{where } l \le u \text{ are some lower and upper integer limits,} \qquad \textbf{(S1)}$$

$$\sum_{i=0}^{n} i = \sum_{i=1}^{n} i = 1 + 2 + \cdots + n = \frac{n(n+1)}{2} \approx \frac{1}{2}n^2 \in \Theta(n^2). \qquad \textbf{(S2)}$$

Note that the formula $\sum_{i=1}^{n-1} 1 = n - 1$, which we used in Example 1, is a special case of formula (S1) for $l = 1$ and $u = n - 1$.

EXAMPLE 1 Consider the problem of finding the value of the largest element in a list of n numbers. For simplicity, we assume that the list is implemented as an array.

ALGORITHM *MaxElement*$(A[0..n-1])$
```
//Determines the value of the largest element in a given array
//Input: An array A[0..n − 1] of real numbers
//Output: The value of the largest element in A
maxval ← A[0]
for i ← 1 to n − 1 do
    if A[i] > maxval
        maxval ← A[i]
return maxval
```

- The obvious measure of an input's size here is the number of elements in the array, i.e., n.
- The operations that are going to be executed most often are in the algorithms **for** loop.
- There are two operations in the loop's body: the comparison $A[i] > maxval$ and the assignment $maxval \leftarrow A[i]$.
- Which of these two operations should we consider basic? Since the comparison is executed on each repetition of the loop and the assignment is not, we should consider the comparison to be the algorithm's basic operation.
- Note that the number of comparisons will be the same for all arrays of size n; therefore, in terms of this metric, there is no need to distinguish among the worst, average, and best cases here.

Let us denote *C(n)* the number of times this comparison is executed and try to find a formula expressing it as a function of size *n*.

The algorithm makes one comparison on each execution of the loop, which is repeated for each value of the loop's variable *i* within the bounds 1 and $n - 1$, inclusive.

Therefore, we get the following sum for *C(n)*:

$$C(n) = \sum_{i=1}^{n-1} 1.$$

This is an easy sum to compute because it is nothing other than 1 repeated $n - 1$ times. Thus,

$$C(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n). \quad \blacksquare$$

EXAMPLE 2 Consider the ***element uniqueness problem***: check whether all the elements in a given array of *n* elements are distinct. This problem can be solved by the following straightforward algorithm.

ALGORITHM *UniqueElements*$(A[0..n-1])$

```
//Determines whether all the elements in a given array are distinct
//Input: An array A[0..n − 1]
//Output: Returns "true" if all the elements in A are distinct
//          and "false" otherwise
for i ← 0 to n − 2 do
    for j ← i + 1 to n − 1 do
        if A[i] = A[j] return false
return true
```

- The natural measure of the input's size here is again *n,* the number of elements in the array.
- Since the innermost loop contains a single operation (the comparison of two elements), we should consider it as the algorithm's basic operation.
- Note, however, that the number of element comparisons depends not only on *n* but also on whether there are equal elements in the array and, if there are, which array positions they occupy.
- We will limit our investigation to the worst case only.

$$C_{worst}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i)$$

$$= \sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i = (n-1) \sum_{i=0}^{n-2} 1 - \frac{(n-2)(n-1)}{2}$$

$$= (n-1)^2 - \frac{(n-2)(n-1)}{2} = \frac{(n-1)n}{2} \approx \frac{1}{2}n^2 \in \Theta(n^2).$$

We also could have computed the sum $\sum_{i=0}^{n-2}(n-1-i)$ faster as follows:

$$\sum_{i=0}^{n-2} (n-1-i) = (n-1) + (n-2) + \cdots + 1 = \frac{(n-1)n}{2},$$

where the last equality is obtained by applying summation formula (S2). Note that this result was perfectly predictable: in the worst case, the algorithm needs to compare all $n(n-1)/2$ distinct pairs of its n elements. ■

EXAMPLE 3 Given two $n \times n$ matrices A and B, find the time efficiency of the definition-based algorithm for computing their product $C = AB$. By definition, C is an $n \times n$ matrix whose elements are computed as the scalar (dot) products of the rows of matrix A and the columns of matrix B:

ALGORITHM *MatrixMultiplication*$(A[0..n-1, 0..n-1], B[0..n-1, 0..n-1])$

```
//Multiplies two square matrices of order n by the definition-based algorithm
//Input: Two n × n matrices A and B
//Output: Matrix C = AB
for i ← 0 to n − 1 do
    for j ← 0 to n − 1 do
        C[i, j] ← 0.0
        for k ← 0 to n − 1 do
            C[i, j] ← C[i, j] + A[i, k] ∗ B[k, j]
return C
```

- We measure an input's size by matrix order n.
- There are two arithmetical operations in the innermost loop here-multiplication and addition-that, in principle, can compete for designation as the algorithm's basic operation. Actually, we do not have to choose between them, because on each repetition of the innermost loop each of the two is executed exactly once. So by counting one we automatically count the other. Still, following a well- established tradition, we consider multiplication as the basic operation.
- Let us set up a sum for the total number of multiplications $M(n)$ executed by the algorithm. (Since this count depends only on the size of the input matrices, we do not have to investigate the worst-case, average-case, and best-case efficiencies separately.)

Total number of multiplications *M(n)* is expressed by the following triple sum:

$$M(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1.$$

Now, we can compute this sum by using formula (S1) and rule (R1) given above. Starting with the innermost sum $\sum_{k=0}^{n-1} 1$, which is equal to n (why?), we get

$$M(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1 = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} n = \sum_{i=0}^{n-1} n^2 = n^3.$$

Mathematical Analysis of Recursive Algorithms:

General Plan for Analysing the Time Efficiency of Recursive Algorithms

1. Decide on a parameter (or parameters) indicating an input's size.
2. Identify the algorithm's basic operation.
3. Check whether the number of times the basic operation is executed can vary on different inputs of the same size; if it can, the worst-case, average-case, and best-case efficiencies must be investigated separately.
4. Set up a recurrence relation, with an appropriate initial condition, for the number of times the basic operation is executed.
5. Solve the recurrence or, at least, ascertain the order of growth of its solution.

EXAMPLE 1 Compute the factorial function *F(n)* = *n*! for an arbitrary nonnegative integer *n*. Since

$$n! = 1 \ldots \ldots (n-1).\ n = (n-1)!\ n \quad \text{for } n \geq 1$$

and 0! = 1 by definition, we can compute *F(n)* = *F* (*n* − 1). *n* with the following recursive algorithm.

ALGORITHM *F*(*n*)
//Computes *n*! recursively
//Input: A nonnegative integer *n*
//Output: The value of *n*!
if $n = 0$ **return** 1
else return $F(n-1) * n$

- For simplicity, we consider *n* itself as an indicator of this algorithm's input size (rather than the number of bits in its binary expansion).
- The basic operation of the algorithm is multiplication; whose number of executions we denote *M(n)*.
- Since the function *F(n)* is computed according to the formula

$$F(n) = F\ (n-1).\ n \text{ for } n > 0,$$

the number of multiplications *M(n)* needed to compute it must satisfy the equality

$$M(n) = \underbrace{M(n-1)}_{\text{to compute } F(n-1)} + \underbrace{1}_{\text{to multiply } F(n-1) \text{ by } n} \quad \text{for } n > 0.$$

Indeed, *M (n − 1)* multiplications are spent to compute *F (n − 1),* and one more multiplication is needed to multiply the result by *n.*

To determine a solution uniquely, we need an ***initial condition*** that tells us the value with which the sequence starts. We can obtain this value by inspecting the condition that makes the algorithm stop its recursive calls:

if $n = 0$ **return** 1.

This tells us two things. First, since the calls stop when $n = 0$, the smallest value of *n* for which this algorithm is executed and hence *M(n)* defined is 0. Second, by inspecting the pseudocode's exiting line, we can see that when $n = 0$, the algorithm performs no multiplications. Therefore, the initial condition we are after is

$M(0) = 0.$

the calls stop when $n = 0$ ⟵ ⟶ no multiplications when $n = 0$

Thus, we succeeded in setting up the recurrence relation and initial condition for the algorithm's number of multiplications $M(n)$:

$$M(n) = M(n-1) + 1 \quad \text{for } n > 0,$$
$$M(0) = 0.$$

From the several techniques available for solving recurrence relations, we use what can be called the ***method of backward substitutions***. The method's idea (and the reason for the name) is immediately clear from the way it applies to solving our particular recurrence:

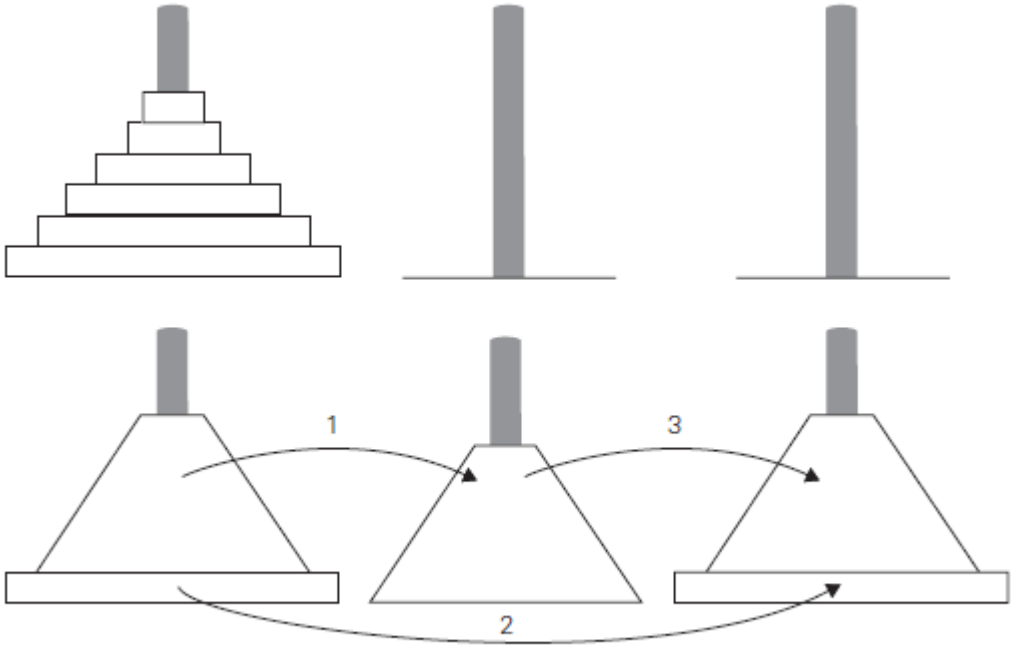
$$\begin{aligned} M(n) &= M(n-1) + 1 && \text{substitute } M(n-1) = M(n-2) + 1 \\ &= [M(n-2) + 1] + 1 = M(n-2) + 2 && \text{substitute } M(n-2) = M(n-3) + 1 \\ &= [M(n-3) + 1] + 2 = M(n-3) + 3. \end{aligned}$$

After inspecting the first three lines, we see an emerging pattern, which makes it possible to predict not only the next line (what would it be?) but also a general formula for the pattern: $M(n) = M(n-i) + i$. Strictly speaking, the correctness of this formula should be proved by mathematical induction, but it is easier to get to the solution as follows and then verify its correctness.

What remains to be done is to take advantage of the initial condition given. Since it is specified for $n = 0$, we have to substitute $i = n$ in the pattern's formula to get the ultimate result of our backward substitutions:

$$M(n) = M(n-1) + 1 = \cdots = M(n-i) + i = \cdots = M(n-n) + n = n.$$

EXAMPLE 2 As our next example, we consider another educational workhorse of recursive algorithms: The ***Tower of Hanoi*** puzzle. In this puzzle, we (or mythical monks, if you do not like to move disks) have n disks of different sizes that can slide onto any of three pegs. Initially, all the disks are on the first peg in order of size, the largest on the bottom and the smallest on top. The goal is to move all the disks to the third peg, using the second one as an auxiliary, if necessary. We can move only one disk at a time, and it is forbidden to place a larger disk on top of a smaller one.



Recursive solution to the Tower of Hanoi puzzle.

- The problem has an elegant recursive solution, which is illustrated in above Figure.
- To move $n>1$ disks from peg 1 to peg 3 (with peg 2 as auxiliary), we first move recursively $n-1$ disks from peg 1 to peg 2 (with peg 3 as auxiliary),
- Then move the largest disk directly from peg 1 to peg 3, and, finally, move recursively $n-1$ disks from peg 2 to peg 3 (using peg 1 as auxiliary).
- Of course, if $n = 1$, we simply move the single disk directly from the source peg to the destination peg.

Let us apply the general plan outlined above to the Tower of Hanoi problem.

The number of disks n is the obvious choice for the input's size indicator, and so is moving one disk as the algorithm's basic operation. Clearly, the number of moves $M(n)$ depends on n only, and we get the following recurrence equation for it:

$$M(n) = M(n-1) + 1 + M(n-1) \text{ for } n > 1.$$

With the obvious initial condition $M(1) = 1$, we have the following recurrence relation for the number of moves $M(n)$:

$$\boldsymbol{M(n) = 2M(n-1) + 1 \text{ for } n > 1,}$$
$$\boldsymbol{M(1) = 1.}$$

We solve this recurrence by the same method of backward substitutions:

$$\begin{aligned} M(n) &= 2M(n-1) + 1 && \text{sub. } M(n-1) = 2M(n-2) + 1 \\ &= 2[2M(n-2) + 1] + 1 = 2^2 M(n-2) + 2 + 1 && \text{sub. } M(n-2) = 2M(n-3) + 1 \\ &= 2^2[2M(n-3) + 1] + 2 + 1 = 2^3 M(n-3) + 2^2 + 2 + 1. \end{aligned}$$

The pattern of the first three sums on the left suggests that the next one will be $2^4 M(n-4) + 2^3 + 2^2 + 2 + 1$, and generally, after i substitutions, we get

$$M(n) = 2^i M(n-i) + 2^{i-1} + 2^{i-2} + \cdots + 2 + 1 = 2^i M(n-i) + 2^i - 1.$$

Since the initial condition is specified for $n = 1$, which is achieved for $i = n - 1$, we get the following formula for the solution to recurrence (2.3):

$$\begin{aligned} M(n) &= 2^{n-1} M(n - (n-1)) + 2^{n-1} - 1 \\ &= 2^{n-1} M(1) + 2^{n-1} - 1 = 2^{n-1} + 2^{n-1} - 1 = 2^n - 1. \end{aligned}$$

ALGORITHM *BinRec*(n)

//Input: A positive decimal integer n
//Output: The number of binary digits in n's binary representation
if $n = 1$ **return** 1
else return *BinRec*$(\lfloor n/2 \rfloor) + 1$

Let us set up a recurrence and an initial condition for the number of additions $A(n)$ made by the algorithm. The number of additions made in computing *BinRec*$(\lfloor n/2 \rfloor)$ is $A(\lfloor n/2 \rfloor)$, plus one more addition is made by the algorithm to increase the returned value by 1. This leads to the recurrence

$$A(n) = A(\lfloor n/2 \rfloor) + 1 \quad \text{for } n > 1.$$

Since the recursive calls end when n is equal to 1 and there are no additions made then, the initial condition is

$$A(1) = 0.$$

$$A(2^k) = A(2^{k-1}) + 1 \quad \text{for } k > 0,$$
$$A(2^0) = 0.$$

Now backward substitutions encounter no problems:

$$
\begin{aligned}
A(2^k) &= A(2^{k-1}) + 1 && \text{substitute } A(2^{k-1}) = A(2^{k-2}) + 1\\
&= [A(2^{k-2}) + 1] + 1 = A(2^{k-2}) + 2 && \text{substitute } A(2^{k-2}) = A(2^{k-3}) + 1\\
&= [A(2^{k-3}) + 1] + 2 = A(2^{k-3}) + 3 && \ldots\\
&\ldots\\
&= A(2^{k-i}) + i\\
&\ldots\\
&= A(2^{k-k}) + k.
\end{aligned}
$$

Thus, we end up with

$$A(2^k) = A(1) + k = k,$$

or, after returning to the original variable $n = 2^k$ and hence $k = \log_2 n$,

$$A(n) = \log_2 n \in \Theta(\log n).$$

In fact, one can prove (Problem 7 in this section's exercises) that the exact solution for an arbitrary value of n is given by just a slightly more refined formula $A(n) = \lfloor \log_2 n \rfloor$. ■

Brute force design technique:

Brute force is a straightforward approach to solving a problem, usually directly based on the problem statement and definitions of the concepts involved.

Selection Sort:

ALGORITHM *SelectionSort*$(A[0..n-1])$
//Sorts a given array by selection sort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
for $i \leftarrow 0$ **to** $n-2$ **do**
  $min \leftarrow i$
  for $j \leftarrow i+1$ **to** $n-1$ **do**
    if $A[j] < A[min]$ $min \leftarrow j$
  swap $A[i]$ and $A[min]$

89	45	68	90	29	34	**17**
17	45	68	90	**29**	34	89
17	29	68	90	45	**34**	89
17	29	34	90	**45**	68	89
17	29	34	45	90	**68**	89
17	29	34	45	68	90	**89**
17	29	34	45	68	89	90

- The analysis of selection sort is straightforward. The input size is given by the number of elements n;
- The basic operation is the key comparison $A[j] < A[min]$.
- The number of times it is executed depends only on the array size and is given by the following sum:

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i).$$

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} (n-1-i) = \frac{(n-1)n}{2}.$$

Thus, selection sort is a $\Theta(n^2)$ algorithm on all inputs.

Bubble Sort:

ALGORITHM *BubbleSort*$(A[0..n-1])$
//Sorts a given array by bubble sort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
for $i \leftarrow 0$ **to** $n-2$ **do**
  for $j \leftarrow 0$ **to** $n-2-i$ **do**
    if $A[j+1] < A[j]$ swap $A[j]$ and $A[j+1]$

89 ? 45 68 90 29 34 17
45 89 ? 68 90 29 34 17
45 68 89 ? 90 ? 29 34 17
45 68 89 29 90 ? 34 17
45 68 89 29 34 90 ? 17
45 68 89 29 34 17 | 90

45 ? 68 ? 89 ? 29 34 17 | 90
45 68 29 89 ? 34 17 | 90
45 68 29 34 89 ? 17 | 90
45 68 29 34 17 | 89 90

etc.

First two passes of bubble sort on the list 89, 45, 68, 90, 29, 34, 17. A new line is shown after a swap of two elements is done. The elements to the right of the vertical bar are in their final positions and are not considered in subsequent iterations of the algorithm.

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=0}^{n-2-i} 1 = \sum_{i=0}^{n-2} [(n-2-i) - 0 + 1]$$

$$= \sum_{i=0}^{n-2} (n-1-i) = \frac{(n-1)n}{2} \in \Theta(n^2).$$

The number of key swaps, however, depends on the input. In the worst case of decreasing arrays, it is the same as the number of key comparisons:

$$S_{worst}(n) = C(n) = \frac{(n-1)n}{2} \in \Theta(n^2).$$

Sequential Search:

ALGORITHM *SequentialSearch2*$(A[0..n], K)$

```
//Implements sequential search with a search key as a sentinel
//Input: An array A of n elements and a search key K
//Output: The index of the first element in A[0..n − 1] whose value is
//        equal to K or −1 if no such element is found
A[n] ← K
i ← 0
while A[i] ≠ K do
        i ← i + 1
if i < n return i
else return −1
```

Another straightforward improvement can be incorporated in sequential search if a given list is known to be sorted: searching in such a list can be stopped as soon as an element greater than or equal to the search key is encountered.

Sequential search provides an excellent illustration of the brute-force approach, with its characteristic strength (simplicity) and weakness (inferior efficiency).

- Clearly, the running time of this algorithm can be quite different for the same list size n.
- In the worst case, when there are no matching elements or the first matching element happens to be the last one on the list, the algorithm makes the largest number of key comparisons among all possible inputs of size n:

$$\boldsymbol{C_{worst}(n) = n.}$$

- The best-case inputs for sequential search are lists of size n with their first element equal to a search key; accordingly, for this algorithm.

$$\mathbf{C_{best}(n) = 1}$$

To analyze the algorithm's average case efficiency, we must make some assumptions about possible inputs of size n.

- The probability of a successful search is equal to p $(0 \le p \le 1)$ and
- The probability of the first match occurring in the i^{th} position of the list is the same for every i.

$$\begin{aligned} C_{avg}(n) &= [1 \cdot \frac{p}{n} + 2 \cdot \frac{p}{n} + \cdots + i \cdot \frac{p}{n} + \cdots + n \cdot \frac{p}{n}] + n \cdot (1-p) \\ &= \frac{p}{n}[1 + 2 + \cdots + i + \cdots + n] + n(1-p) \\ &= \frac{p}{n}\frac{n(n+1)}{2} + n(1-p) = \frac{p(n+1)}{2} + n(1-p). \end{aligned}$$

Brute-Force String Matching:

ALGORITHM *BruteForceStringMatch*$(T[0..n-1], P[0..m-1])$

```
//Implements brute-force string matching
//Input: An array T[0..n − 1] of n characters representing a text and
//       an array P[0..m − 1] of m characters representing a pattern
//Output: The index of the first character in the text that starts a
//        matching substring or −1 if the search is unsuccessful
for i ← 0 to n − m do
    j ← 0
    while j < m and P[j] = T[i + j] do
        j ← j + 1
    if j = m return i
return −1
```

A brute-force algorithm for the string-matching problem is quite obvious: align the pattern against the first m characters of the text and start matching the corresponding pairs of characters from left to right until either all the m pairs of the character's match (then the algorithm can stop) or a mismatching pair is encountered. In the latter case, shift the pattern one position to the right and resume the character comparisons, starting again with the first character of the pattern and its counterpart in the text. Note that the last position in the text that can still be a beginning of a matching substring is $n – m$ (provided the text positions are indexed from 0 to $n − 1$). Beyond that position, there are not enough characters to match the entire pattern; hence, the algorithm need not make any comparisons there.

```
N O B O D Y _ N O T I C E D _ H I M
N O T
  N O T
    N O T
      N O T
        N O T
          N O T
            N O T
              N O T
```

Example of brute-force string matching. The pattern's characters that are compared with their text counterparts are in bold type.

- The worst case, the algorithm makes ***m (n − m + 1)*** character comparisons, which puts it in the ***O(nm)*** class.
- For a typical word search in a natural language text, however, we should expect that most shifts would happen after very few comparisons (check the example again). Therefore, the average-case efficiency should be considerably better than the worst-case efficiency. Indeed, it is: for searching in random texts, it has been shown to be linear, i.e., $\Theta(n)$.